Part I
Allocation

Constructor calls are allocation:

```
(define (interp)
  (type-case ExprD expr-reg
    ...
    [lamD (body-expr)
      (begin
        (set! v-reg (closV body-expr env-reg))
        (continue))]
    ...))

(define (continue)
  ...
  [addSecondK (r env k)
    (begin
      (set! expr-reg r)
      (set! env-reg sc)
      (set! k-reg (doAddK v-reg k))
      (interp))]
  ...
```

Deallocation

Where does free go?

(define (continue)
  ...
  [doAddK (v1 k)
    (begin
      (set! v-reg (num+ v1 v-reg))
      (free k-reg) ; ???
      (set! k-reg k)
      (continue))]
  ...
  [doAppK (fun-val k)
    (begin
      (set! expr-reg (closV-body fun-val))
      (set! env-reg (cons v-reg
          (closV-env fun-val)))
      (set! k-reg k)
      (free fun-val) ; ???
      (interp))]
  ...
  )
Deallocation

\[ \text{doAddK (v1 k)} \]
\begin{verbatim}
(begin
  (set! v-reg (num+ v1 v-reg))
  (free k-reg) ; ???
  (set! k-reg k)
  (continue))
\end{verbatim}

- Without \texttt{let/cc}, this free is fine, because the continuation can’t be referenced anywhere else

- A continuation object is always freed as \texttt{(free k-reg)}, which is why many language implementations use a stack
Deallocation

\[ \text{doAppK (fun-val k)} \]
\[
\text{(begin}
\text{(set! expr-reg (closV-body fun-val))}
\text{(set! env-reg (cons v-reg}
\text{ (closV-env fun-val))))}
\text{(set! k-reg k)}
\text{(free fun-val) ; ???}
\text{(interp))}]\]

- This free is not ok, because the closure might be kept in a environment somewhere
- Need to free only if no one else is using it...
Code and Data

An **object** is any record allocated during **interp** and **continue**

Assume that expressions are allocated “statically”

- **compile** uses **code-malloc1**, etc.
- Only try to free objects allocated during **interp** and **continue**
Part 2
Reference Counting

**Reference counting:** a way to know whether an object has other users

- Attach a count to every object, starting at 0
- When installing a pointer to an object (into a register or another object), increment its count
- When replacing a pointer to an object, decrement its count
- When a count is decremented to 0, decrement counts for other objects referenced by the object, then free
Reference Counting

Top boxes are the registers \texttt{k-reg}, \texttt{v-reg}, etc.

Boxes in the blue area are allocated with \texttt{malloc}
Reference Counting

Adjust counts when a pointer is changed...
Reference Counting

... freeing an object if its count goes to 0
Reference Counting

Same if the pointer is in a register
Reference Counting

Adjust counts after frees, too...
Reference Counting

... which can trigger more frees
Reference Counting in an Interpreter

...  
[lamC (body-expr)
  (begin
    (ref- v-reg)
    (set! v-reg
     ; must ref+ env:
     (closV body-expr env-reg))
    (ref+ v-reg)
    (continue))]
...

[doAppK (fun-val k)
  (begin
    (set! fae-reg (closV-body fun-val)) ; code is static
    (ref- env-reg)
    (set! env-reg
     ; must ref+ each arg:
     (cons v-reg (closV-env fun-val)))
    (ref+ env-reg) ; => ref+ on v-reg
    (ref+ k)
    (ref- k-reg) ; => ref- on fun-val and k
    (set! k-reg k)
    (interp))]
Reference Counting And Cycles

An assignment can create a cycle...
Reference Counting And Cycles

Adding a reference increments a count
Reference Counting And Cycles

Lower-left objects are inaccessible, but not deallocated

In general, cycles break reference counting
Part 3
Garbage Collection

**Garbage collection:** a way to know whether an object is *accessible*

- An object referenced by a register is *live*
- An object referenced by a live object is also live
- A program can only possibly use live objects, because there is no way to get to other objects
- A garbage collector frees all objects that are not live
- Allocate until we run out of memory, then run a garbage collector to get more space
Garbage Collection Algorithm

• Color all objects \textit{white}

• Color objects referenced by registers \textit{gray}

• Repeat until there are no gray objects:
  • Pick a gray object, $r$
  • For each white object that $r$ points to, make it gray
  • Color $r$ \textit{black}

• Deallocate all white objects
Garbage Collection

All objects are marked white
Garbage Collection

Mark objects referenced by registers as gray
Garbage Collection

Need to pick a gray object

Red arrow indicates the chosen object
Garbage Collection

Mark white objects referenced by chosen object as gray
Garbage Collection

Mark chosen object black
Garbage Collection

Start again: pick a gray object
Garbage Collection

No referenced objects; mark black
Garbage Collection

Start again: pick a gray object
Garbage Collection

Mark white objects referenced by chosen object as gray
Garbage Collection

Mark chosen object black
Garbage Collection

Start again: pick a gray object
Garbage Collection

No referenced white objects; mark black
Garbage Collection

No more gray objects; deallocate white objects

Cycles **do not** break garbage collection
Part 4
Two-Space Copying Collectors

A **two-space** copying collector compacts memory as it collects, making allocation easier.

**Allocator:**

- Partitions memory into **to-space** and **from-space**
- Allocates only in **to-space**

**Collector:**

- Starts by swapping **to-space** and **from-space**
- Coloring gray $\Rightarrow$ copy from **from-space** to **to-space**
- Choosing a gray object $\Rightarrow$ walk once though the new **to-space**, update pointers
Two-Space Collection

Left = from-space
Right = to-space
Two-Space Collection

Mark gray = copy and leave forward address
Two-Space Collection

Choose gray by walking through to-space
Two-Space Collection

Mark referenced as gray
Two-Space Collection

Mark black = move gray-choosing arrow
Two-Space Collection

Nothing to color gray; increment the arrow
Two-Space Collection

Increment the gray-choosing arrow
Two-Space Collection

Referenced is already copied, use forwarding address
Two-Space Collection

Choosing arrow reaches the end of to-space: done
Two-Space Collection

Right = from-space
Left = to-space
Part 5
Two-Space Collection on Vectors

• Everything is a number:
  ◦ Some numbers are immediate integers
  ◦ Some numbers are pointers

• An allocated object in memory starts with a tag, followed by a sequence of pointers and immediate integers
  ◦ The tag describes the shape
Two-Space Vector Example

- 26-byte memory (13 bytes for each space), 2 registers
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer

Register 1: 7  
Register 2: 0

From: 1 75 2 0 3 2 10 3 2 2 3 1 4
Two-Space Vector Example

- 26-byte memory (13 bytes for each space), 2 registers
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer

<table>
<thead>
<tr>
<th></th>
<th>Register 1: 7</th>
<th>Register 2: 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>From:</td>
<td>1 75 2 0 3 2 10 3 2 2 3 1 4</td>
<td></td>
</tr>
<tr>
<td>Addr:</td>
<td>00 01 02 03 04 05 06 07 08 09 10 11 12</td>
<td></td>
</tr>
</tbody>
</table>
Two-Space Vector Example

- 26-byte memory (13 bytes for each space), 2 registers
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer

<table>
<thead>
<tr>
<th>Register 1: 7</th>
<th>Register 2: 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>From:</td>
<td>Addr:</td>
</tr>
<tr>
<td>1 75 2 0 3 2 10 3 2 2 3 1 4</td>
<td>00 01 02 03 04 05 06 07 08 09 10 11 12</td>
</tr>
<tr>
<td></td>
<td>^ ^ ^ ^ ^ ^ ^ ^ ^</td>
</tr>
</tbody>
</table>
Two-Space Vector Example

- 26-byte memory (13 bytes for each space), 2 registers
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer

<table>
<thead>
<tr>
<th>Register 1: 7</th>
<th>Register 2: 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>From:</td>
<td>From:</td>
</tr>
<tr>
<td>Addr:</td>
<td>Addr:</td>
</tr>
<tr>
<td>1 75 2 0 3 2 10 3 2 2 3 1 4</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>00 01 02 03 04 05 06 07 08 09 10 11 12</td>
<td></td>
</tr>
</tbody>
</table>
Two-Space Vector Example

- 26-byte memory (13 bytes for each space), 2 registers
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer

<table>
<thead>
<tr>
<th>Register 1: 0</th>
<th>Register 2: 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>From:</td>
<td>Addr:</td>
</tr>
<tr>
<td>1    75   2   0   3   2   10  99   0   2   3   1   4</td>
<td></td>
</tr>
<tr>
<td>00  01  02  03  04  05  06  07  08  09  10  11  12</td>
<td></td>
</tr>
<tr>
<td>^   ^   ^   ^   ^   ^   ^   ^   ^   ^   ^   ^   ^</td>
<td></td>
</tr>
<tr>
<td>To:</td>
<td></td>
</tr>
<tr>
<td>3    2    2   0   0   0   0   0   0   0   0   0   0</td>
<td></td>
</tr>
<tr>
<td>^</td>
<td></td>
</tr>
</tbody>
</table>
Two-Space Vector Example

• 26-byte memory (13 bytes for each space), 2 registers
  ○ Tag 1: one integer
  ○ Tag 2: one pointer
  ○ Tag 3: one integer, then one pointer

<table>
<thead>
<tr>
<th>Register 1: 0</th>
<th>Register 2: 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>From:</td>
<td>Addr:</td>
</tr>
<tr>
<td>99 3 2 0 3 2 10 99 0 2 3 1 4</td>
<td>00 01 02 03 04 05 06 07 08 09 10 11 12</td>
</tr>
<tr>
<td></td>
<td>^ ^ ^ ^ ^</td>
</tr>
<tr>
<td>To:</td>
<td></td>
</tr>
<tr>
<td>3 2 2 1 75 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>^</td>
<td></td>
</tr>
</tbody>
</table>
Two-Space Vector Example

- 26-byte memory (13 bytes for each space), 2 registers
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer

<table>
<thead>
<tr>
<th>Register 1: 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>From: 99 3 99 5 3 2 10 99 0 2 3 1 4</td>
</tr>
<tr>
<td>Addr: 00 01 02 03 04 05 06 07 08 09 10 11 12</td>
</tr>
<tr>
<td>^ ^ ^ ^ ^</td>
</tr>
<tr>
<td>To: 3 2 5 1 75 2 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>
Two-Space Vector Example

- 26-byte memory (13 bytes for each space), 2 registers
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer

<table>
<thead>
<tr>
<th>Register 1: 0</th>
<th>Register 2: 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>From:</td>
<td>Addr:</td>
</tr>
<tr>
<td>99 3 99 5 3 2 10 99 0 2 3 1 4</td>
<td>00 01 02 03 04 05 06 07 08 09 10 11 12</td>
</tr>
<tr>
<td></td>
<td>^ ^ ^ ^ ^</td>
</tr>
<tr>
<td>To:</td>
<td></td>
</tr>
<tr>
<td>3 2 5 1 75 2 0 0 0 0 0 0 0 0 0</td>
<td>^ ^</td>
</tr>
</tbody>
</table>
Two-Space Vector Example

- 26-byte memory (13 bytes for each space), 2 registers
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer

<table>
<thead>
<tr>
<th>Register 1: 0</th>
<th>Register 2: 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>From: 99 3 99 5 3 2 10 99 0 2 3 1 4</td>
<td></td>
</tr>
<tr>
<td>Addr: 00 01 02 03 04 05 06 07 08 09 10 11 12</td>
<td></td>
</tr>
<tr>
<td>To: 3 2 5 1 75 2 3 0 0 0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

^ ^ ^ ^